



Mamba Image Library Coding Rules and Standards

Nicolas BEUCHER
Serge BEUCHER

February 12, 2011

Contents

1	Introduction	4
2	Policy	4
2.1	A Mathematical Morphology library	4
2.2	Simple yet Fast	4
2.3	Portable	4
2.4	... and Free	4
3	Programming	5
3.1	Languages	5
3.2	Rules for C	5
3.3	Rules for Python	5
3.3.1	General rules for simplicity and readability	6
3.3.2	Rules for operators using 'grid' and/or 'se' as arguments	6
3.3.3	Rules for input and output images	8
3.3.4	Rules for 'edge' argument in user-defined operators	9
4	Documentation	9
4.1	In code	9
4.2	Other documents	10
5	Testing	10
6	Licensing	10
7	Other contributions	11

List of Figures

standards, n.:

The principles we use to reject other people's code.

1 Introduction

Mamba is an open-source library and any contribution is welcome. In order to help programmers, users and would be contributors, this document presents the basic policy ruling the development of Mamba. It is our vision, as its creators, of what is Mamba and where we want it to go in the future. Anyone who wish to participate (by giving ideas, suggestions or code) should read at least the policy section [2](#).

This document also provides a set of rules, guidelines and general standards to use when developing Mamba. Its intended audience is programmers contributing to the Mamba library. These rules are not intended to be followed blindly but are meant to make it easier for the various programmers to understand the code they are not directly responsible for but still need to interact with. As a programmer for Mamba, you should take care to fulfil these rules and ultimately, if you have a good reason for breaking them, to correctly explain and document your reason.

Documentation contributions and licensing aspects are also covered by this document.

2 Policy

As you may already know, Mamba objective is to be a fast, simple, portable and free (as in free speech but also as in free beer, this being the case only because no one on Earth could afford it if we had decided to sell it) mathematical morphology library. To cover this purpose, a basic policy was decided by its creators.

2.1 A Mathematical Morphology library

Mamba is a mathematical morphology library only. It basically means that there is no convolution, Fast Fourier Transform and such in it. Only algorithms related to mathematical morphology may be present in it.

This policy is one of the founding aspect of Mamba. The objective is to prevent Mamba from dispersing itself into aspects that are not related to its original purpose. We believe that there is already enough good libraries out there to compute images using other techniques than mathematical morphology and thus that there is no need for Mamba to include them.

2.2 Simple yet Fast

Fast and simple can appear somehow contradictory as complex algorithms may deliver faster performances. However, what we mean here is to make sure Mamba is simple to use. As such, we believe that it is important not to have to wait a very long time to get the result of your algorithm. Mamba is meant to be used in research and education where you are not always sure of your idea. Being able to rapidly evaluate the soundness of an algorithm is also what will make Mamba simple to use. In other words, the faster Mamba is, the simpler it becomes to try new ideas with it.

It seems also important for us to provide tools, inside Mamba, helping users to "visualize" their ideas and thus be able to easily assess their worthiness. As a result, Mamba comes with a set of tools to fulfil this objective, the most visible being the capability to see manipulated images in live. These tools are a very important part of the library (equally in our eyes to the core functions used in computations). Again, the objective is to provide a simple framework for testing and trying new ideas in mathematical morphology.

2.3 Portable

Mathematical morphology algorithms variety and diversity make it somehow stupid to try to restrain their use to a very limited set of computers and devices (or so we think). Thus the effort towards portability.

2.4 ... and Free

Last but not least, as we hope Mamba will become a huge success and help spread knowledge and use of mathematical morphology, we believe it is important not to restrain its future users with a complex or obscure license. Similarly, we think that a commercial license is inappropriate for some of the intended audience of this library (students, university...) and too complex for us to handle with others (industry, ...). Thus we decided to license Mamba under a slightly modified version of the X11 license (also known as the MIT license). This is one of the least restrictive free license. Any contributor to Mamba will have to make sure his contribution is licensed under a similar license (see [6](#)).

3 Programming

3.1 Languages

- Mamba low level is written in C.
- Mamba high level is written in Python.

This language choice is meant to cover the Mamba objective of being a fast portable and easy to use library.

C choice for low-level answers the speed requirement and partly covers the portability. Indeed, Mamba is meant to be portable to embedded systems. Of course, the part written in C must be compilable on various environments (Windows, Unix ...) and for various processors (Pentium, Core 2 Duo, ARM, ...).

Python choice is here to ensure easiness of use. As a high level language, it makes it easier to develop programs and algorithms without taking care of memory management, OS portability ... Other languages could be used for high level but currently all the high level functionalities are written in Python. Of course, if you have the time and the need to develop a high level interface in another language, feel free to do so. However, maintaining multiple high level interfaces in different languages may prove too much hassle so your high level interface may not be integrated to the official Mamba distribution.

3.2 Rules for C

Three words can be used to sum up the philosophy of rules/standards applying to Mamba C code.

- simplicity
- readability
- portability

Simplicity means that your code must not try to answer all the problems or to take into account all the possible situations. You should leave the complexity to the high level interface (Where generally it is much easier to handle the real life situations).

Readability is a vague notion. Mainly, you have to make sure your code is understandable. Comments, coherent naming, and so on are strongly advised. More specific rules are :

- Use of english in comments is mandatory.
- Function descriptions in comments use Doxygen style (see the documentation section below for more details regarding documentation). At least all the exported functions must have a description.
- Indentations are done using 4 spaces (no tabs, they are ugly because they messed up when changing the editor).
- Functions and variables that are exported (visible by the exterior) should begin with "MB_".

Portability means that you should always take care to write your program so that it will run on the greatest number of machines and equipments. Of course this is not always feasible (particularly if you are going very low level). Anyway, it simply means that if, for example, you write an algorithm using SSE instructions of modern Intel/AMD processors, you should also include your algorithm in standard C code (even if that means it's excruciatingly slow).

3.3 Rules for Python

When coding in Python, you must take great care to make your code and the various functionalities you are implementing as simple to use as possible.

The rules presented here aim at defining good practices in the design of new Mamba operators in order that these operators be as general and usable as possible. Although these rules are not compulsory, you are advised to follow them if you wish to share your work. Presently, these rules concern mainly the use of the very important arguments, edge, structuring element (se) and grid. The use of input and output images is also considered.

3.3.1 General rules for simplicity and readability

The Python API is meant to be used easily by users regardless of their proficiency at programming (of course they need to have a basic idea of what they are doing).

Complexity is handled in the Python interface. Most of the time, C functions only deal with image data and their depth. The Python code is in charge of calling the appropriate C function depending on the image depth it is dealing with. Thus the same function is used for binary, greyscale and 32-bit images in Python and corresponds to three functions in C (one for each depth). Complexity should only be handled in C if this makes computations go significantly faster.

As for comments, you need to document each function that is meant to be public with docstring. You should also begin your internal functions with a "_" that will tell Python that it's an internal function (the same goes for global variables).

Every non internal function must provide a docstring explaining what it does, its arguments and its output. Keep in mind that this docstring will be automatically extracted to build the Python/Mamba reference documentation.

3.3.2 Rules for operators using 'grid' and/or 'se' as arguments

The following five rules address particularly the use of the two arguments 'grid' (grid used by the operator, hexagonal or square) and 'se' (structuring element possibly used by the operator). As 'se' is always defined in association with a specific grid, possible conflicts are at stake if these two arguments are used in an antagonistic way inside the Mamba/Python scripts defining the operators.

Rule n° 1

When creating an operator or wrapping a C function, make sure that the grid, edge and structuring element that may be given as argument have a default value available when calling the Python function thus making it easier for interactive sessions (by reducing the number of mandatory arguments).

For example, the following C function:

```
MB_errcode MB_InfNbb(MB_Image *src ,
                    MB_Image *srcdest ,
                    unsigned int nbrnum ,
                    unsigned int count ,
                    enum MB_grid_t grid ,
                    enum MB_edgemode_t edge);
```

is wrapped in Python by the function :

```
def infNeighbor(imIn , imInout , nb , count , grid=DEFAULT_GRID , edge=FILLED):
```

Information regarding edge and grid have default values in the Python module and thus do not need to be defined every time.

The same rule applies for structuring element, for example:

```
def dilate(imIn , imOut , n=1 , se=DEFAULT_SE , edge=mamba.EMPTY):
```

Rule n° 2

If 'se' is passed as an argument in the operator, as 'se' is always associated with a grid (for instance, SQUARE3X3 is defined on the square grid), internal operators using a grid as argument must use this grid in their argument list. This 'grid' argument can be passed by means of the method getGrid() of the structuringElement class.

Example:

buildOpen definition in module openClose.py (commented script):

```
def buildOpen(imIn , imOut , n=1 , se=mC.DEFAULT_SE)
    """
    Performs an opening by reconstruction operation on image 'imIn' and puts the
    result in 'imOut'. 'n' controls the size of the opening. 'se' is passed as
    default argument.
    """
    imWrk = mamba.imageMb(imIn)
    mamba.copy(imIn , imWrk)
    mC.erode(imIn , imOut , n , se=se)
    # 'se' is used by the erosion in the first step. You can use any structuring
```

```

# element you wish to achieve this operation.
mC.build(imWrk, imOut, grid=se.getGrid())
# However, as you use a build operator in the second step and as this operator
# depends on the grid in use, you must pass to it the grid associated with
# 'se'. This is done through the statement grid=se.getGrid() which assigns
# to 'grid' the grid on which 'se' is defined.

```

Rule n° 3

If 'grid' is used in the operator argument list, internal operators needing a structuring element 'se' as argument must explicitly define this 'se'. Make sure that 'se' is compatible with the 'grid' in use.

Example:

This example is not very useful, it is just to illustrate the rule...

```

def myOperator(imIn, imOut, grid=mamba.DEFAULT_GRID)
    """
    Performs an erosion of 'imIn' and puts the result in 'imOut'. 'grid' is
    passed as default argument. If 'grid' is hexagonal, the erosion must use an
    hexagon and a square if 'grid' is square.
    """
    if grid == mamba.HEXAGONAL:
        se = mC.HEXAGON
    else:
        se = mC.SQUARE3X3
    # The structuring element 'se' is defined according to the grid in use.
    erode(imIn, imOut, se=se)

```

If you do not define explicitly the structuring element used by the erosion, DEFAULT_SE will be used. However, you have no idea of the status of this default structuring element (it could be DIAMOND for instance whereas the grid in use is hexagonal...).

You may argue that the definition of HEXAGON or SQUARE3X3 may also be changed by the user. This is true and this is the reason why it is not wise to modify the definition of standard structuring elements.

A lot of operators use what is called “an elementary structuring element” which is in fact the size 1 structuring element defined on the grid: if the grid is hexagonal, it corresponds to HEXAGON (in the standard definition) and to SQUARE3X3 if we deal with a square grid. In this situation, if you want to be sure to use an unmodified structuring element, you may proceed this way instead:

```

def myOperator(imIn, imOut, grid=mamba.DEFAULT_GRID)
    """
    Performs an erosion of 'imIn' and puts the result in 'imOut'. 'grid' is
    passed as default argument. If 'grid' is hexagonal, the erosion must use an
    hexagon and a square if 'grid' is square.
    """
    se = structuringElement(mamba.getDirections(grid), grid):
    # The structuring element 'se' is still defined according to the grid in
    # use. Its definition uses directly the characteristics of the grid, that is
    # the list of all its directions (including direction 0) and the grid itself
    # which is necessarily associated to 'se'.
    erode(imIn, imOut, se=se)

```

Rule n° 4

Using at the same time 'se' and 'grid' in an operator argument list is strictly forbidden. This practice is at best redundant (if 'se' and 'grid' are compatible), at worst dangerous and inconsistent.

It is obvious that, if this rule is not enforced, this will lead sooner or later to a “grid and structuring element mess”.

Rule n° 5

It is possible that no argument ('grid' or 'se') be passed to an operator. This is allowed provided that no conflict is generated by this means. To achieve this, make sure that all internal operators use only either 'grid' or 'se'. If it is not the case (for instance, there exists in the definition script at least one operator requesting 'grid' and another one requesting 'se'), this situation is likely to produce conflicts and errors.

If no argument is passed, the operator will use DEFAULT_SE or DEFAULT_GRID defined outside its definition space (with a possible risk of conflicts).

Example:

```

def alternateFilter(imIn, imOut, n):

```

```

"""
Performs an alternate filter operation of size 'n' on image 'imIn' and puts
the result in 'imOut'. If 'openFirst' is True, the filter begins with an
opening, a closing otherwise.
"""
if openFirst:
    mC.open(imOut, imOut, n)
    mC.close(imOut, imOut, n)
else:
    mC.close(imOut, imOut, n)
    mC.open(imOut, imOut, n)
# Neither open nor close need 'grid' in their arguments, but only 'se'.
# Therefore, as there is, in the definition of alternateFilter, no other
# operator requesting 'grid', no conflict occurs and the structuring element
# used by open and close will be DEFAULT_SE.

```

Although this rule is admitted, it is not wise in practice to apply it. Indeed, to be applied without risk, this rule states that all internal operators need only one argument, 'grid' or (exclusive) 'se', it is therefore more efficient and safe to systematically pass this argument in the operator argument list. This avoids later difficulties if the operator itself is used in the definition of more complex functions.

Thus, the following definition of 'alternateFilter' is safer and usable inside another definition:

```

def alternateFilter(imIn, imOut, n, se=mamba.DEFAULT_SE):
    """
    Performs an alternate filter operation of size 'n' on image 'imIn' and puts
    the result in 'imOut'. If 'openFirst' is True, the filter begins with an
    opening, a closing otherwise.
    """
    if openFirst:
        mC.open(imOut, imOut, n, se=se)
        mC.close(imOut, imOut, n, se=se)
    else:
        mC.close(imOut, imOut, n, se=se)
        mC.open(imOut, imOut, n, se=se)

```

These five rules are not mandatory and do not constitute a dogma. However, using them will avoid some nasty problems during your first steps with Mamba. They allow also a better applications sharing.

Most of the time, 'grid' and 'se' are passed in the argument list as default arguments (se=mC.DEFAULT_SE and grid=mamba.DEFAULT_GRID). This is very handy when you use Mamba operators in terminal/console mode to test ideas and to concatenate these operators in order to solve a problem. In this case, you generally work in a chosen environment, you know which grid you are using (the grid which is defined as default grid) and you control totally your structuring elements. For instance, if your grid is hexagonal, it is likely that your default structuring element is set to HEXAGON.

This interesting simplification when in interactive mode, is unacceptable when you define a new operator if you want to share it with other users. In this case, you do not control the environment anymore and you must make sure that this operator will be used according to your wishes. Therefore, it is of the outmost importance to correctly manage the 'grid' and 'se' arguments in your definition.

3.3.3 Rules for input and output images

You can define with Mamba new operators using one or more input image(s) and putting some results in one or more output image(s). An important programming rule is that your operators should be defined in such a way that the same image may be used both as input and output. This means that you must be aware to avoid to replace your input image by any output image prematurely. Indeed, if you do not take care of this rule, it is likely that, in most cases, no error will occur. However, your result will certainly be wrong.

Example:

```

# First approach (bad one).

def myOper(imIn, imOut):
    """
    This operator computes a very simple morphological gradient of imIn and
    puts the result in imOut.
    """

```

```

imWrk = imageMb(imIn)
dilate(imIn, imOut)
erode(imIn, imWrk)
sub(imOut, imWrk, imOut)

```

In this example, if `imIn` and `imOut` are identical, although no error is detected, the final result will be incorrect since, in the `erode` operator, `imIn` has been replaced by its dilation.

Correct approach.

```

def myOper(imIn, imOut):
    """
    This operator computes very simple morphological gradient of imIn and
    puts the result in imOut.
    """
    imWrk = imageMb(imIn)
    dilate(imIn, imWrk)
    erode(imIn, imOut)
    sub(imWrk, imOut, imOut)

```

In this case, `imIn` may possibly be overwritten after the erosion with no harmful consequences regarding the final result.

Here again, enforcing this rule makes your operators user friendly and their sharing is made easier. This rule can be implemented very easily by using intermediary working images in your definition and by copying the input `imIn` into the working image at the beginning of this definition.

3.3.4 Rules for 'edge' argument in user-defined operators

Contrary to 'grid' and 'se', it is generally neither necessary nor wise to pass 'edge' in your operators arguments list.

There is no default value in Mamba for 'edge'. The reason of this is that 'edge' is always used to impose specific behavior and properties to some operators. It is the case for instance with geodesic operators. It is well known that non extensive geodesic operators need that 'edge' be defined as 'FILLED'. Obviously, for these operators, the edge setting is defined inside the operator definition and not at all in its argument list.

In fact, 'edge' allows to define two different contexts for your transformation depending on the way you consider the outer space, that is the space outside your image window.

If you consider that the outer space is empty, you are then in an euclidean context. In this configuration, 'edge' is defined as 'EMPTY' and the main consequence of this choice is that outer 'black' pixels may have some influence on the result of your transformation for inside pixels, in particular if your transformation is anti-extensive. Note that you could equally define the outer space as 'FILLED' in this context (and Mamba allows it without problem). However, this configuration is seldom used in practice, being considered as "unnatural" (we prefer to imagine the outer space completely empty rather than totally filled!).

If you consider that you have no idea of the content of the outer space and, moreover, that you don't care of it, you are working in a geodesic context. Your image window is considered as a geodesic space and you do not have to take the outside pixels into account in your operator. This can be achieved by setting 'edge' accordingly in your definition. If an extensive operator requesting the setting of 'edge' is called in the definition, 'edge' should be set to 'EMPTY'. Conversely, if the operator is anti-extensive, 'edge' should be set to 'FILLED'.

It is important to keep in mind the fact that an empty edge is NOT equivalent to an euclidean context and, conversely, that a filled edge is NOT equivalent to a geodesic context. This is the main reason why 'edge' should not be passed in your operator argument list.

The only case where this rule can be by-passed is when you know that the operator you are defining can be used in both contexts and that it produces a legitimate result (but possibly different). This is true for instance for the basic morphological operators: dilations, erosions, openings, closings, thinnings and thickenings. So, when defining another implementation of these basic transforms (this is perfectly allowed), it is nevertheless wise (if not compulsory) to pass 'edge' in the operator argument list.

4 Documentation

4.1 In code

As was explained in the programming section, documentation related to code must take two forms :

For Python code, use docstring.

For C code, use Doxygen style.

You should indeed know that part of the actual documentation is created automatically using these forms of documentation. The whole documentation must be written in english.

4.2 Other documents

All the other documents existing to date were created using Latex. Mamba comes with a specific Latex style that allows creating an homogeneous set of documents (with header, code listing style, etc... coherent between documents).

If you have an idea of documentation, we strongly advise you to use Latex along with the Mamba style (which can be found in the source). To use it, you simply need to create a directory `texmf/tex/latex` in your `$HOME` or wherever your Latex distribution may find it and add the files `mamba.sty` and `mamba_logo_white.png` in it. We prefer Latex because it makes it easier to track modifications inside our versioning repository (Latex being text format and not binary).

For those of you who do not have Latex or who do not wish to use it, we can accept documents in other formats provided you also give along a PDF version of them (making sure that way that the document will be readable by anyone).

And of course, whatever the format you choose, make sure your name appears in the document.

5 Testing

Mamba is tested to ensure that it works properly and that it implements correctly the mathematical morphology algorithms it uses.

Testing is performed in a specific environment. We tried to avoid verification with test images because they are not easy to handle (often a lot of images to store) and although they correctly reveal errors, they cannot be used to identify and correct them. Another problem with test images is the production of the expected result image and the insurance that it is correct.

There are two levels of testing:

- **BASIC C LEVEL OPERATORS/FUNCTIONS** : They are tested and verified as extensively as possible to ensure that they are working properly and do not crash. The testing is performed by calling their wrapping Python function. Algorithm soundness, edge effects, grid effects, image depth acceptance, proper error handling are tested for every C function.
- **MATHEMATICAL OPERATORS AND OTHER HIGH-LEVEL PYTHON FUNCTIONS** : These are tested less thoroughly as this is not a realistic task. Basic algorithmic verification is performed. The idea is to cover all the Python code (100% code coverage) to make sure there is no typo.

Although we take great care of producing a bug free library, there is no way for us to guarantee that. Tests are likely to have some blind spot. They are mainly here to ensure a bottom line quality level and to prevent any regression. If you have written some operators/functions and would like to add it into the official Mamba release, we would be grateful if you could provide us with some tests to verify them.

6 Licensing

For code contribution, make sure your work is licensed under a similar license than the one covering Mamba. Here is a reminder of the license:

Copyright (c) <2009>, <Your name here>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

Except as contained in this notice, the names of the above copyright holders shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without their prior written authorization.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Any similar license is appropriate (see X11 license, BSD license). Make sure your license is compatible with GNU GPL and that it has NO copyleft obligations (GPL is a copyleft license). If your license has a copyleft obligation, we will not be able to add your code to the Mamba source. However an optional package/module can be created that will let the user decide if the copyleft obligation is a problem for her/him or not (see the `mambaRealttime` for an example of this situation).

For documentation contributions, make sure that your work license allow us to distribute it freely.

7 Other contributions

There are lot of things you could do for Mamba, even if your are not a programmer or a writer.

First of all, you can give us feedback regarding the way you use Mamba. Any comment, criticism or suggestion is welcome and will be taken into consideration (as long as it does not infringe our policy).